Chair for Algebraic and Logical
Foundations of Computer Science
TU Dresden

Winter Term 2021/2022

Logic Programming Engineering – Assignment 11

Dr.-Ing. Sascha Klüppelholz

Exercise 11.1 [Operator specification]

a) Define a predicate `operator/1` that allows for enumerating the predefined operators. The definition should rely on `current_op/3`.

Solution:

```
operator(Operator) :-
    current_op(_,_,Operator).
```

b) Use `current_op/3` to check the properties (precedence and type) for a few predefined operators of your own choice.

Solution:
no solution provided.

c) Define a new binary operator `&` with precedence 950 (which is slightly less than the comma which has 1000) and `xfy` for the tpye. The operator should just make a call (using `call/1`) on two goals. Use again `current_op/3` to check the properties of the newly defined operator.

Solution:

```
% define & as operator
:- op(950,xfy,&).

% explain semantics
GoalA & GoalB :- call(GoalA), call(GoalB).
```

d) Start with the following simple fact: `is_bigger_than(elephant,mouse)` and make `is_bigger_than` an operator with precedence 300 and `xfx` for the type. You should now be able to use infix notation rather than prefix notation:

Solution:

```
% new predicate
is_bigger_than(elephant,mouse).

% becomes an operator with P=300 and A=xfx
:- op(300,xfx,is_bigger_than).
```

```
?- elephant is_bigger_than mouse = is_bigger_than(elephant,mouse).
true.
```

Exercise 11.2 [Truth table of Boolean functions]

The goal of this exercise is to print the truth table of an arbitrary Boolean function.

Solution[1]:

```
:- op(1000,xfy,'and').
:- op(1000,xfy,'or').
:- op(900,fy,'not').

and(false,false,false).
and(false,true,false).
and(true,false,false).
and(true,true,true).

or(false,false,false).
or(false,true,true).
or(true,false,true).
or(true,true,true).

not(false,true).
not(true,false).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

find_vars(X,V) :- find_vars(X,[],V).

find_vars(N,V,V) :- member(N,[false,true]),!.

find_vars(X,Vin,Vin)      :- atom(X), member(X,Vin), !.
find_vars(X,Vin,[X|Vin]) :- atom(X).

find_vars(X and Y,Vin,Vout) :- find_vars(X,Vin,Vtemp),
                               find_vars(Y,Vtemp,Vout).

find_vars(X or Y,Vin,Vout) :-  find_vars(X,Vin,Vtemp),
                               find_vars(Y,Vtemp,Vout).

find_vars(not X,Vin,Vout) :-   find_vars(X,Vin,Vout).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

initial_assign([],[]).
initial_assign([_|R],[false|S]) :- initial_assign(R,S).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

[1] based on the solution by J.R. Fisher (https://www.cpp.edu/~jrfisher/www/prolog_tutorial/2_13.html)

```
successor(A,S) :- reverse(A,R),
                  next(R,N),
                  reverse(N,S).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

next([false|R],[true|R]).
next([true|R],[false|S]) :- next(R,S).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

evaluate_expression(N,_,_,N) :- member(N,[false,true]).

evaluate_expression(X,Vars,A,Val) :- atom(X),
                                     lookup(X,Vars,A,Val).

evaluate_expression(X and Y,Vars,A,Val) :- evaluate_expression(X,Vars,A,VX),
                                           evaluate_expression(Y,Vars,A,VY),
                                           and(VX,VY,Val).

evaluate_expression(X or Y,Vars,A,Val) :-  evaluate_expression(X,Vars,A,VX),
                                           evaluate_expression(Y,Vars,A,VY),
                                           or(VX,VY,Val).

evaluate_expression(not X,Vars,A,Val) :-   evaluate_expression(X,Vars,A,VX),
                                           not(VX,Val).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

lookup(X,[X|_],[V|_],V).
lookup(X,[_|Vars],[_|A],V) :- lookup(X,Vars,A,V).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tt(E) :- find_vars(E,V),
         reverse(V,Vars),
         initial_assign(Vars,A),
         write('␣␣'), write(Vars), write('␣␣␣␣'), writeln(E),
         writeln('---------------------------------------'),
         write_row(E,Vars,A),
         writeln('---------------------------------------'),!.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

write_row(E,Vars,A) :- write('␣␣'), write(A), write('␣␣␣␣␣␣␣␣'),
                       evaluate_expression(E,Vars,A,V), write(V), nl,
                       (successor(A,N) -> write_row(E,Vars,N) ; true).
```

Exercise 11.3 [Call of an arbitrary function]

Write a predicate `map/3` that allows for applying an arbitrary function to all elements of a list. E.g., given the two predicates

```
neg(A,B)    :- B is -A.
square(A,B) :- B is A**2.
```

for negating and squaring one element, the `map/3` predicate should behave as indicated in the examples below:

```
?- map(neg,[-1,2,-3,-4,5],Result).
Result = [1, -2, 3, 4, -5].

?- map(square,[-1,2,-3,-4,5],Result).
Result = [1, 4, 9, 16, 25].
```

Solutions:

```
map(_,[],[]).

map(FunctionName,[H|T],[NH|NT]):-
    Function=..[FunctionName,H,NH],
    call(Function),
    map(FunctionName,T,NT).

neg(A,B)    :- B is -A.
square(A,B) :- B is A**2.
```

Exercise 11.4 [Term analysis]

The goal of this exercise is to analyze a given term T and provide a user information on the type and properties of T.

Provide a predicate `analyse_term/1` that prints information on the given term T and recognizes the following types and properties of T:

- T is an atom or ground term (cf. `atom/1` and `ground/1`)
- T is of type string (cf. `string/1`)
- T is compound or atomic (cf. `atomic/1` and `compound/1`)
- T is a variable/ not a variable (cf. `var/1` and `nonvar/1`)
- T is a number of a certain type (cf. `number/1`, `integer/1`, `float/1`, and `rational/3`)
- T is cyclic/ acyclic (cf. `cyclic/1` and `acyclic/1`)
- T is callable (cf. `callable/1`)

Solution:

```
analyse_term(T) :-
    compound(T), !, writef("%t␣is␣a␣compound␣term.\n",[T]),
    compound_name_arity(T,F,Arity),
    writef("found␣functor:␣%t/%t\n",[F,Arity]),
    compound_name_arguments(T,F,Args),
    analyse_terms(Args).
```

```
analyse_term(T) :-
    analyse_terms([T]).

analyse_terms([]) :- !.

analyse_terms([H|T]) :-
    compound(H), !, writef("%t␣is␣a␣compound␣term.\n",[H]),
    H =.. [F|Args],
    length(Args,Arity),
    nl,
    writef("found␣operator:␣%t/%t\n",[F,Arity]),
    append(Args,T,S),
    analyse_terms(S).

analyse_terms([H|T]) :-
    nl,
    (var(H) ->          writef("found␣unbound␣variable:␣%t\n",[H]); true),
    (nonvar(H),         writef("found␣non-variable:␣%t\n",[H]); true),

    (number(H),         tab(2), writef("%t␣is␣a␣number.\n",[H]); true),
    (integer(H),        tab(2), writef("%t␣is␣an␣integer.\n",[H]); true),
    (float(H),          tab(2), writef("%t␣is␣a␣float.\n",[H]); true),
    (rational(H,N,D),   tab(2), writef("%t␣is␣a␣rational.",[H]); true),

    (atom(H),           tab(2), writef("found␣an␣atom:␣%t\n",[H]); true),

    (blob(H,T),         tab(2), writef("found␣blob:␣%t\n",[H]); true),

    (string(H),         tab(2), writef("found␣a␣string:␣%t\n",[H]); true),

    (atomic(H),         tab(2), writef("%t␣is␣atomic.\n",[H]); true),
    (callable(H),       tab(2), writef("%t␣is␣callable.\n",[H]); true),
    (ground(H),         tab(2), writef("%t␣is␣a␣ground␣term.\n",[H]); true),
    (cyclic_term(H),    tab(2), writef("%t␣is␣a␣cyclic␣term.\n",[H]); true),
    (acyclic_term(H),   tab(2), writef("%t␣is␣an␣acyclic␣term.\n",[H]); true), !,
    analyse_terms(T).
```

Exercise 11.5 [Arbitrary function evaluation]

Provide a predicate `func/1` that allows to compute the value of an arbitrary arithmetic function given as a parameter of the predicate. In case that the function contains free (i.e., unbound) variables, the user should be asked to provide a numeric constant value.

Solution:

```
func(T) :-
  (compound(T) ->
    functor(T,F,A),
    writef("found␣functor:␣%t␣with␣%t␣arguments.\n",[F,A]),
    T =.. [_|Terms];

    Terms = [T]
```

```
  ),
  bind_the_unbound(Terms),
  FX is T,
  writef("%t = %t\n",[T,FX]).

bind_the_unbound([]) :- !.

bind_the_unbound([H|T]) :-
    compound(H), !,
    compound_name_arguments(H,F,Args),
    writef("found operator: %t\n",[F]),
    append(Args,T,S),
    bind_the_unbound(S).

bind_the_unbound([H|T]) :-
    repeat,
    (var(H) ->
      writef("found unbound variable: %t\n",[H]), read(H), number(H);
    true),
    bind_the_unbound(T).
```